Event Documentation

5 October 2006

`itos/src/dsp/event` contains source files for the following programs:

1. dsp_evtdsp The C/Motif program currently used with TCW software.
2. EvtdspApp A java applet/application for viewing event messages. It implements only a subset of dsp_evtdsp functions, omitting things like opening connections to more than one source.
3. evtforward A java background application that forwards event messages from a source to multiple consumers. Typical sources are dsp_evtdsp or another instance of evtforward. Typical consumers are dsp_evtdsp, EvtdspApp or another instance of evtforward.

Message formats Describes messages used by all three programs as well as several other ITOS programs such as dsp_evtlog.

1 dsp_evtdsp

Dsp_evtdsp is a process which can receive event messages from dsp_evtlog on stdin. It can also receive messages over sockets from other instances of dsp_evtdsp and from evtforward and send messages from dsp_evtlog to other dsp_evtdsp and to evtforward.

The user can open views on events from any of the sources mentioned above and can specify which message types are displayed in each view.

Command line arguments
Message formats
Overall organization

1.1 dsp_evtdsp command line arguments

-help

When "-help" is present on the command line, dsp_evtdsp prints information about its command line arguments and exits.

-port "num"

Instructs dsp_evtdsp to open port "num" as its server port instead of the default, 6066.

"view"

Instructs dsp_evtdsp to open a view named "view", displaying events from the local dsp_evtlog. (Note that there is no hyphen before the view name.)

This view name can be followed by a comma and a comma-separated list of event specifications. Event specifications include:

+* => Add all events.
-* => Delete all events.
+N => Add event number N.
-N => Delete event number N.

By default, all events are displayed in the view.

-getView "source" "view"

Instructs dsp_evtdsp to open a view named "view" displaying event messages from "source". If the source is another instance of dsp_evtdsp using the default port number, "source" can simply be the name of the node where that dsp_evtdsp is running. Otherwise "source" should use the format "node>port".

As described above, "view" may be followed by a comma and a comma-separated list of event specifications.

The command line may contain any number of -getview directives.

1.2 Overall organization of dsp_evtdsp

1.2.1 External connections

When dsp_evtdsp is run with dsp_evtlog, the script that starts them usually pipes the output of dsp_evtlog to dsp_evtdsp. That output consists of event messages, which dsp_evtdsp reads on stdin. Before the main application loop is started, function ReadEventMessages is installed as the stdin message handler.

Dsp_evtdsp can also exchange event messages via sockets with other processes, including those running on other machines. In order to insulate dsp_evtdsp from network problems, a child process is spawned which handles all socket connections with other processes. Dsp_evtdsp communicates with its child, named dsp_remote, over pipes. Sending an event message to a remote dsp_evtdsp involves sending the message to the local child over a pipe, which sends it to the remote child over a socket, which sends it to its parent dsp_evtdsp over a pipe.

When the user wants to get a view of events from a remote dsp_evtdsp, she performs two steps:

1. Establishes a connection with the remote dsp_evtdsp. This is done by bringing up the manager window, selecting view/new display... and entering the name of the remote dsp_evtdsp.

   dsp_evtdsp reacts to this by sending an idOpenDisplay message to its child dsp_remote instructing it to open a socket to the remote dsp_evtdsp. (message formats describes messages between dsp_evtdsp and dsp_remote.)
2. Requests a view. This is done by bringing up the manager window, selecting the tab for the remote dsp_evtdsp, selecting view/get view... and entering a name for the view.

   dsp_evtdsp reacts to this by sending an idGetView to dsp_remote, which forwards the message to the remote dsp_evtdsp.

1.2.2 Major data structures

1.2.2.1 struct OpenDisplay

Each OpenDisplay structure corresponds to a page in the manager window. There is one for the local process and one for every other process with which this process has a socket connection. There is also one for local devices like a printer.

A new OpenDisplay struct is created when the user selects view/"new display..." in the manager window. A new OpenDisplay struct is also created when a user on another process gets a view of events from this process or sends a view to this process.

Some fields in OpenDisplay:

- `char *name;` The name for the process appearing on the manager tab.
- `char *ownName;` This field is needed because other processes can refer to this process by a variety of strings. If the string used by the remote process is different from the `name` field in the OpenDisplay for this process, that string is saved here and is used in the "sending process" field in messages to that process.

- `View *viewsTo;` A linked list of views owned by the remote process showing events from this process. The OpenDisplay struct for the local process puts all of its views into this list.
- `View *viewsFrom;` A linked list of views owned by this process showing events from the remote process.
- `OpenDisplay *next;` Used to link all OpenDisplay structures into a single list pointed to by the global `firstDisplay`.
- `OpenDisplay *nextFrom;` Used to link structures of processes which send at least one view to this process. The global `fromRemote` points to this list.
- `OpenDisplay *nextTo;` Used to link structures of processes which receive at least one view from this process. The global `toRemote` points to this list.

  This field is also used to link the list of structures for processes which have no views at the moment. The global emptyRemote points to this list. The toRemote and emptyRemote lists are mutually exclusive so there's no problem using the same field to link the two lists.

If event messages are flowing both ways between this process and a remote process, the OpenDisplay struct for the remote process will be in both the toRemote and fromRemote lists and will have views in both its viewsTo and viewsFrom lists.

1.2.2.2 struct View

There is one View structure for every view displayed by this process and for each view it sends to some other process.

Some fields in View:

- `char *name;` The view's name. If the view is displayed locally, this name is in the view's window header.
- `XmString str;` Contains the same string as `name`. Used to put the display's name into the manager page.
- `OpenDisplay *display;` The OpenDisplay that this view belongs to.
- `View *next` Used to link the view into one of its OpenDisplay's lists.

2 EvtdspApp

EvtdspApp was written primarily to be used as a java applet. It allows a user to open views of events from an event source on the host from which it was downloaded. Class EvtdspButton is derived from class Applet and is designed to appear in an html file. It displays "Start" and "Quit" buttons and a text area.

When a user clicks "Start", a thread is started which executes class EvtdspApp, the class which implements a subset of dsp_evtdsp's functionality. EvtdspApp can also be run as a standalone java application.

Overview gives a high-level introduction to classes and threads.
Html support describes what should be in an html file to run the applet. Parameters are described there as well.
Application EvtdspApp describes how to run EvtdspApp as an application.
Classes describes all major classes and their methods.
Puzzles describes a few wierd things in the code possibly caused by the author's lack of java knowledge.

EvtdspApp is created from source files:

'`EvtdspButton.java`' Defines classes used only by the applet, not the application.

'`MsgBuffer.java`' Defines class MsgBuffer, which handles communications with other processes.

'`EvtUtil.java`' Defines assorted utility classes which handle color and integer strings.

'`EvtdspApp.java`' The main file, defines the classes that implement all other functionality of EvtdspApp.

2.1 Overview of EvtdspApp

2.1.1 EvtdspButton

The `init` and `start` methods of class EvtdspButton are called when the html page is displayed. Init creates two buttons, "Start" and "Quit", and a text area. The "Quit" button is not active at this time. The `start` method insures that the main application class (`EvtdspApp`) is accessable and is a thread.

Only one of the buttons is enabled at any time. Method `actionPerformed` is called when the active button is clicked. If that is the "Start" button, an `EvtdspApp` thread is created and started. When the "Quit" button is clicked, the `EvtdspApp` thread is killed.

2.1.2 EvtdspApp

The `run` method of class EvtdspApp is called whether this program is executing as an applet or an application. It creates startup windows, starts an EventThread to read messages from the event source, and waits for the user to kill the program. Its `destroy` method closes sockets, streams and windows.

2.1.3 EventThread

One instance of EventThread is started by EvtdspApp. Its `run` method simply waits for event messages over the socket from the event source process. When an event message arrives, it is passed to the `eventMsg` method of the manager window.

When the user makes certain changes, such as adding or deleting a view or changing the events displayed in a view, messages must be sent to the event server process. Methods for sending those messages are defined in this class, but they run in the Java monitor thread.

If this thread finds that it has lost contact with the event source process, it goes into a loop in which it tries to reconnect periodically. If it is successful, it requests all currently active views from the server.

2.1.4 MgrWindow

This class implements the manager window. Most of its methods are called in response to user actions in the manager window. An exception is method `eventMsg`, which is called by the EventThread when an event message arrives. This method appends the text of the new event to the views specified in the message.

2.1.5 Views

Every event view is an object of class View. Its visible components are a text area and possibly scroll bars. The text area is an object of class CanvasTextArea. These objects behave much like Java TextArea objects except that:

The program has control over whether the scroll bars are visible. This is useful in implementing autoscrolling.

The program can distinguish clicks in the scroll bars from clicks in the text area.

The program maintains control over how much text is saved so a user can scroll back to it.

The MgrWindow object maintains a list of the current views in an object of class ViewList. Methods in this class add views to the list and delete views from the list. Those methods execute in the Java monitor thread as a result of user actions. Other methods traverse the current list of views. Those are called when an event is received and must be written to the views which want it. Those methods execute in the event thread. Since multiple threads access this list, all methods are synchronized.

2.1.6 Event message handling

This section summarizes what happens when a message arrives over the socket from the event source process.

1. Method `EventThread.run`, which was blocked waiting for input, wakes and determines that the input is an event message. It then calls `EventThread.processEvent`.
2. `EventThread.processEvent` removes the first two strings from the message just received. Those strings give the names of the sending and receiving processes. The remainder of the message, which begins with the event text, is passed to `MgrWindow.eventMsg`.

3. `MgrWindow.eventMsg` looks at the strings in the message following the event text. They are the names of the views where the message should be displayed. It passes the message text to the `appendText` method of each of those views.
4. `appendText` appends the new message to its message queue and decides whether the new message changes what is displayed in the view. If autoscrolling is on, the new message is always displayed at the bottom of the window. Otherwise the new message changes the view only if it forces a displayed line to be deleted or if the window is so large that all text in the queue is visible. If the view is changed, `repaint` is called.
5. The call to `repaint` causes `CArea.paint` to be called. It simply calls `CanvasTextArea.paintText`.
6. `paintText` determines which block of lines from the view's text queue will be written to the window, creates a BlockOfLines object, and then uses it to access the lines in the queue and draw them in the window.

All of these functions execute in the EventThread thread.

2.1.7 Threads

To summarize, three threads are active when this program is run as an applet:

1. The EvtdspApp thread. It creates the manager window, starts the event thread, and creates any startup views. It then suspends itself and is resumed when either:

   The user clicks the `"Quit"` button in the manager window.

   The user clicks evtdspButton's `"Quit"` button.

   After that resumption, this thread cleans up and exits.
2. The eventThread, which receives event messages over a socket and distributes them to views. All methods in class EventThread run in this thread except `newView`, `newFilter` and `deleteView`. (Those methods are called in response to user actions and send messages over the socket to the event source process.) In addition, MgrWindow.evetMsg, ViewList.getView and CanvasTextArea.appendText execute in this thread.
3. The java monitor thread. All methods not mentioned above run in this thread. This includes most methods in MgrWindow and View.

2.2 Html support for EvtdspApp

In an html file, the applet code should be given as `"EvtdspButton.class"`.

Parameter names recognized by EvtdspApp and their values:

- PORT = An integer port number. The applet will connect to this port and request event messages. The default is 6066, the port of the server socket opened by dsp_evtdsp. (evtforward opens 6067 as its server port.)
- TEXT_FONT = The name of the font to be used in text components, including the event-display window. The default is Dialog. Other fonts available to applets are Helvetica, TimesRoman, Courier and Symbol. Applications have many more fonts.

- BUTTON_FONT = The name of the font used everywhere else, including on buttons. The default is Helvetica.
- TEXT_FONT_SIZE = An integer font size for the text font. The default is 14.
- BUTTON_FONT SIZE = An integer font size for the button font. The default is 14.
- VIEW_FG = The color used in view foregrounds. Value can be specified as a color name, an ITOS color code (code_0, etc.) or three comma-separated integers giving rgb values. The default is "white".
- VIEW_BG = The color used in view backgrounds. Values accepted are the same as with VIEW_FG. The default is "0,0,140".
- VIEWi where "i" is an integer. The value should specify one startup view. If there are n VIEW parameters, they must be named VIEW1, VIEW2, ..., VIEWn. The value string consists of the following fields, separated by blanks or tabs:
  1. The view name.
  2. An event specification, using dsp_evtdsp filter format. By default, all events are displayed.
  3. The number of character columns in the view. The default value is 80.
  4. The number of text rows in the view. The default value is 6.
  5. The number of rows in the view which can be seen by scrolling. The default is 500.
  6. Foreground color. Formats and default value are described above with VIEW_FG.
  7. Background color. Formats and default value are described above with VIEW_FG.
  8. A boolean specifying whether the view should be visible at startup. The default is true.
  9. The number of pixels between the left side of the view and the left side of the screen. Negative numbers are ignored by the program. Zero seems to be ignored by Motif.
  10. The number of pixels between the top of the view and the top of the screen. Negative numbers are ignored by the program. Zero seems to be ignored by Motif.

  Only the name field is required. If a field is omitted, all following fields should be omitted too. "def" is legal in all fields except the name field and gives the default value.

  Since blanks serve as field separators, no field can contain a blank.

An html file could look like this:

```
<HTML>
<HEAD>
<TITLE> Event Display </TITLE>
</HEAD>
<BODY>
<APPLET CODE="EvtdspButton.class" WIDTH=400 HEIGHT=200>
<PARAM NAME=PORT VALUE=6066>
<PARAM NAME=VIEW1 VALUE="stol_events -*,+15,+16,+17,+18,+19,+26,+27 45 10 def black cy
<PARAM NAME=VIEW2 VALUE="events">
```

```
</APPLET>
</BODY>
</HTML>
```

2.3 Running EvtdspApp as an application

The following are examples of lines which can be typed at a shell prompt to start EvtdspApp as an application:

```
java EvtdspApp -SERVER sunflower
java EvtdspApp -SERVER sunflower -PORT 6067
java EvtdspApp -SERVER sunflower -VIEW1 "stol_events -*,+15,+16,+17,+18,+19"
```

The `-SERVER` argument is required. It specifies the address of the socket from which EvtdspApp will receive events.

All other command line arguments are identical to the html parameters. Parameter names must be preceded by a hyphen and must be entered using upper case. Each parameter value must consist of a single string.

2.4 Classes of EvtdspApp

BlockOfLines is used when painting views.
CanvasTextArea Implements the text area and scroll bars of a view window.
CArea Implements the text area of CanvasTextArea.
CmdParams Processes parameters from either the command line or an html file.
ErrWindow A window for displaying errors.
EventMsg A single string in a TextQueue.
EventThread Communicates with the event server.
EvtdspApp The main application class.
EvtdspButton The main applet class.
EvtGlobal A class defining various defaults.
FilterList Linked list of FilterViews.
FilterView Window for changing a view's filter.
MgrWindow Implements the manager window.
StartupView One startup view.
StartupViewList The list of all startup views.
TextQueue Implements the queue of messages for a single View.
View Implements one view.
ViewDialog Queries the user for a new view name.
ViewList A linked list of View objects.
ViewPopup The View popup menu.

2.4.1 Class BlockOfLines

When a view is redrawn, the painting method needs a block of contiguous messages from the view's text queue. This is complicated by the fact that lines may be added to and deleted from the queue during the painting process.

In the solution chosen here, the painting method requests a block from the text queue before it begins painting. The request specifies the block relative to the most recent message. The text queue responds to the request by storing the information it needs to identify the block in a small object of this class and returning that object. The painting method then repeatedly calls the queue's `nextLine` method and passes this object to that method.

TextQueue stores its queue of event messages in a fixed-length array. Its member `tail` is the offset in the array where the next message will be placed. The most recent message is stored at tail - 1, the message before that at tail - 2, and so on. When tail gets to the end of the array, it wraps around to zero.

Data members:

```
int newOffset; // Offset in the TextQueue array of the newest
               // line in the block. This is the line which will
               // be returned by the next call to "nextLine".
int oldOffset; // Offset of the oldest line in the block.
int prevTail;  // The value of TextQueue's tail pointer during the
               // previous call to "nextLine". This is used to
               // detect cases where so many lines were added to
               // the queue between calls to "nextLine" that none of
               // the requested lines remain in TextQueue.
int blockNum;  // The value of TextQueue's blockNum when this object
               // was created. Used to tell if a newer block exists.
               // If a newer block exists, painting is stopped on
               // the older block.
```

Objects of this class are created and manipulated by two methods in TextQueue, requestBlock and nextLine. A rule that they both observe is that TextQueue's tail pointer is never in a block. Specifically, that means

```
if(oldOffset <= newOffset)
   // A simple block
   tail <= oldOffset or tail > newOffset
else
   // The block wraps around 0 in the TextQueue array.
   tail <= oldOffset and tail > newOffset
```

If a call to requestBlock tries to create a block containing tail, or if tail moves so it is within a block, the block's `oldOffset` member is set equal to tail, so tail is no longer in the block.

Methods:

```
boolean simple()
```

This method returns true if the block does not wrap around zero, that is, if newOffset >= oldOffset.

```
boolean contains(int i)
```

This method returns true if "i" is in the block. If "i" equals oldOffset, it is not considered to be in the block.

2.4.2 class CanvasTextArea

```
class CanvasTextArea extends Panel implements AdjustmentListener
```

An object of this class implements the visible parts of a single view. The main parts are a text area, implemented by a CArea object, and optionally two java Scrollbars.

Data members:

```
TextQueue text;     // All the text which can be seen by scrolling
                     // the view.
CArea textArea;     // Object that displays the visible part of text.
Scrollbar vbar;      // Vertical scroll bar. Is null when text is
                     // autoscrolling (so the scrollbar is hidden).
Scrollbar hbar;      // Horizontal scroll bar.
int leftCol = 0;   // Number of pixel columns scrolled to the left of
                     // the window.
int canvasPixRows; // Number of pixel rows currently in textArea.
int bottomPixRow;  // Number of pixel rows from the current bottom of
                     // textArea to the top of the oldest line.
                     // Used only when a vertical scrollbar is visible.
int charHeight;     // Height of a character in pixels.
int charWidth;       // Width of 'M' in pixels. Used in horizontal scrolls.
int descHeight;     // Height of a character descender in pixels.
FontMetrics fm;      // Metrics for the font.
Color fg;           // The default foreground color.
View parent;        // The View this object belongs to.
Font font;          // The font being used.

// Constants used in calls to the constructor.
final static boolean makeHScroll = true;
final static boolean noHScroll = false;
final static boolean makeVScroll = true;
final static boolean noVScroll = false;
```

Every View object has one of these objects as its `textArea` data member. At one time `textArea` was implemented as a java TextArea object. A **CanvasTextArea** object gives the functionality of a TextArea object and in addition:

1. Scroll bars can be displayed or hidden at any time. That is done here when the user selects "Autoscroll" from the view's popup menu.
2. Mouse clicks in the text area can be distinguished from clicks in the scroll bars. This is important in knowing when to display the view's popup menu.
3. The displayed text can be treated as a queue of specified length. Old text is discarded when the queue overflows.

```
CanvasTextArea(View parent, int rows, int cols, int maxRows,
         boolean hBar, boolean vBar)

CanvasTextArea(View parent, int rows, int cols, int top, int left,
         int maxRows, boolean hBar, boolean vBar, String fg, String bg)
```

```
CanvasTextArea(View parent, int rows, int cols, int maxRows,
        boolean hBar, boolean vBar, String fg, String bg, Font font)
```

```
void constructor(View parent, int rows, int cols, int top, int left,
       int maxRows, boolean hBar, boolean vBar, String fgStr, String bgStr,
       Font font)
```

The three constructors substitute defaults for any values not specified and then call `constructor`. (Default color values are specified in EvtGlobal.)

`rows` and `cols` specify the number of text rows and columns in the window.

`top` and `left`, if both are non-negative, specify the position of the top left corner of the view in pixels.

`maxRows` is the maximum number of rows of text that will be saved before discarding the oldest text.

`hbar` and `vbar` specify whether horizontal scroll bars should be displayed initially.

`fgStr` and `bgStr` are strings giving foreground and background colors.

`font` is the font to be used for displaying text.

```
void paintText(Graphics g)
```

This method is called when it is time to redraw the text area of the window. Text is drawn starting at the bottom of the window.

Two local variables are used:

1. `i` is an integer giving the index in `text` of the line of text currently being drawn. The last line added to `text` has index 0. (`text` is a TextQueue object. Its `getLine` method is used here.)
2. `base` is an integer giving the Y pixel coordinate of the bottom line of text in the window.

Algorithm:

```
if(vbar is null)
   // We are autoscrolling.
   i = 0
   base = descHeight above the bottom of the window.
else
   Use text.length() and bottomPixRow to set i to the index of
      the bottom line. It may be only partially visible.
   Set base
endif

While(There is still room in the window
and there is more text to draw)
   Get text line i.
   if(the line has a non-default background)
      Draw a rectangle in its background color.
   endif
   if(the line's foreground color is different from the current color)
      Set the foreground color.
   endif
```

```
      Draw the line of text at -leftcol, base.
      Subtract charHeight from base.
      Add one to i.
   endwhile
```

boolean **appendText**(String str)

This method is called by View.`appendText` when a new event message is received from the event server. `str` is the text of the new message. This method adds it to this view.

`str` is passed to text.`append` so it is appended to the text queue. Subsequent actions:

```
if(autoscrolling)
   Just repaint.
else if(this string changes the length of the text queue)
   if(the text does not fill the window)
      Increment bottomPixRow.
      Repaint.
   else
      Modify the maximum value of the scrollbar.
   endif
else
   // The oldest line was just removed from the queue.
   if(That oldest line was not visible)
      Move the scrollbar.
   else
      Repaint.
   endif
endif
```

Finally, the value returned by `text.append()` is returned. That value is true if `str` contained a BEL character.

void **sizeChanged**()

This method is called by View.ComponentResized when the user resizes the window.

It sets `canvasPixRows` by calling `textArea`'s `getSize` method.

If the vertical scroll bar is visible, its values are reset. The goal is to keep the same line displayed at the bottom of the window.

void **toggleAutoscroll**()

If `vbar` is currently null, this method creates a vertical scrollbar. Otherwise, it removes the scrollbar and sets `vbar` to null.

public void **adjustmentValueChanged**(AdjustmentEvent e)

This method implements the AdjustmentListener interface. It is called when the user adjusts either of the scrollbars.

When the vertical scrollbar is adjusted, `bottomPixRow` is updated, and when the horizontal bar is adjusted, `leftCol` is updated. In both cases, the text area is then repainted.

2.4.3 class CArea

```
class CArea extends Canvas
```

This class implements the text area part of CanvasTextArea.

Data members:

```
Dimension minSize;      // The initial size of the area.
CanvasTextArea parent; // The CanvasTextArea this object is
                        // contained in.
```

The constructor is:

```
CArea(int rows, int cols, Color bg, CanvasTextArea parent)
```

`rows` and `cols` specify the initial size of the canvas in pixels. They are used to set `minSize`. `bg` specifies the default background color. `parent` specifies the object whose `paintText` method is called by `paint`.

The constructor creates an instance of the inner class `MAdapter` and registers it as the mouse listener for this CArea.

```
class MAdapter extends MouseAdapter
```

This is an inner class of `CArea`. It defines a single method:

```
public void mouseClicked(MouseEvent e)
```

This method is called for mouse clicks in the canvas area. If the third mouse button was clicked and the popup menu is not currently visible, the popup menu is displayed at the location of the click.

If the popup menu is already visible, it is hidden.

Three other methods are defined:

1. preferredSize() returns minumumSize().
2. minimumSize() returns data member `minSize`.
3. paint(Graphics g) calls parent.paintText(g).

2.4.4 class CmdParams

```
class CmdParams
```

This class allows a program to access startup parameters through a common interface whether they come from the command line or an html file. (That is, whether the program is run as an application or applet.)

When parameters are given on a command line, it is assumed that the parameter name is preceded by a hyphen and that the parameter value is a single string which does not begin with a hyphen.

```
//***  Data members  ***
String[] args; // Used only when run as an application.
Applet applet; // Used only when run as an applet.
final String emptyParam = ""; // Sometimes returned by getValue.
public CmdParams(String[] args)
public CmdParams(Applet applet)
```

These constructors set the data member passed to them. The first constructor is used by applications, the second by applets.

```
String getValue(String paramName)
```

This method returns the value associated with the parameter named "paramName". Null is returned if "paramName" is not defined. If "paramName" appears on the command line with no value `emptyParam`, an empty string, is returned.

If `applet` is non-null, this method looks for the parameter's value in the html file by calling `applet.getParameter`. Otherwise this method looks through `args` for the parameter.

2.4.5 class ErrWindow

```
class ErrWindow extends Frame implements ActionListener
```

One object of this class is instantiated by EvtdspApp for displaying errors. It contains a 2-line text area and an OK button.

Data members:

```
Button okButton;
TextArea textArea;
```

Text is written directly to `textArea` by `EvtdspApp.nonFatalError`().

The following functions are defined:

1. A constructor which does not take any arguments and which creates components for the text area and button.

   It creates an instance of inner class DWAdapter and registers it as the window listener.
2. **actionPerformed** implements the `ActionListener` interface. It responds to clicks on the OK button by hiding this window.
3. Inner class

   ```
   class DWAdapter extends WindowAdapter
   ```

   Defines the single method:

   ```
   public void windowClosing(WindowEvent event)
   ```

   which detects user clicks on the frame's "Close" menu item and hides this window.

2.4.6 class EventMsg

An object of this class contains the text of an event message with foreground and background colors.

Data Members:

```
String text;   // Event text, without formatting prefix.
Color fg;      // Foreground color.
Color bg;       // background color.
                // These colors are left null if the event message
                // does not specify them, or if they are specified as
                // 0 (default background color) or 7 (default
                // foreground color).
```

Class TextQueue implements a queue of these objects.

There are no method members in this class.

2.4.7 class EventThread

```
class EventThread extends Thread
```

One instance of this thread is created by EvtdspApp.run. Methods of this class connect to the event server and exchange messages with it. The `run` method waits for messages from the event server and passes them to `MgrWindow.eventMsg()`

Method `connectToEvents`, defined here, is called by `EvtdspApp.run` before this class's `run` is called because error-handling for a socket failure is easier in that thread.

Three other methods, `newView`, `deleteView` and `newFilter`, are defined in this class because they write to the event socket, but they don't run in this thread. They are called by functions in the manager window class.

Data members:

```
EvtdspApp app;           // Pointer to the EvtdspApp object.
Socket eventSocket;      // Socket to events.
String host;      // Address of eventSocket.
int port;                // Port number of eventSocket.
DataInputStream is;      // Input stream for reading events.
DataOutputStream os; // Output stream for writing to the
                         // the event server.
MsgBuffer msgBuffer; // Used by methods running in this thread.
MsgBuffer sendBuffer;// Used by other threads.

final int waitMillis = 5000; // When we are not in contact with the
                         // event server, we sleep this many milliseconds
                         // between attempts to reconnect.
String sendDspName;       // This string is used as sendProc in messages
                         // to the event server. See Evtdsp messages.
```

```
EventThread(String host, int port, EvtdspApp app)
```

This constructor sets data members `host`, `port` and `app` to its arguments. It then creates a name from this applet consisting of the string "applet" followed by the time. Finally, it allocates buffers `msgBuffer` and `sendBuffer`.

```
String connectToEvents()
```

This method connects to the server socket at `host`, `port`. It sets data members `eventSocket`, `is` and `os`. If all goes well, null is returned. Otherwise an error string is returned.

This method is called by `EvtdspApp.run` before this thread is started. This is done because if the method is not successful in connecting to the event server, `EvtdspApp` wants to know about it immediately, since it cannot proceed farther.

```
public void run()
```

Algorithm for this method:

```
while(the user has not quit)
   Wait for a message on eventSocket.
   if(the user quit)
      return
      else if(The socket is closed)
         Call method reconnect.
      else if(some other error)
         Pass an error string to app.nonFatalError().
      else if(The message is an event message)
         Call method processEvent to handle it.
      else
         Error "Strange message type received"
      endif
   end repeat
```

This thread is stopped when some other thread calls `app.wake`, which sets `userQuit` to true.

`protected void` **processEvent**`()`

When this method is called, `msgBuffer` contains an idEvent message just received from the event server. This method deletes strings from `msgBuffer` so that the next string is the event text. `msgBuffer` is then passed to `MgrWindow.eventMsg`.

`private void` **reconnect**`()`

This method is called when it appears that we have lost contact with the event server. It tries to reconnect to the server every `waitMillis` milliseconds until it is successful. After it reconnects, it requests all views that are currently open.

Algorithm:

```
repeat
   Close the socket to the event server.
   Write an error message saying contact has been lost.
   repeat
      Try to reconnect.
      if(unsuccessful) Sleep for waitMillis.
   until(successfully reconnected or the user quit)
   Request all views from the event server.
until(view-request messages are successfully sent or the user quits)
Hide the error message window.
```

`boolean` **requestViews**`()`

When this method is called, we have just reconnected to the event server after a break in connection. This function sends a request message for all current views. It returns true if all goes well, false otherwise.

`boolean` **newView**`(String name, long filter)`
`boolean` **newView**`(String name, long filter, MsgBuffer buf)`

The first method is called by the manager window thread when the user requests a new view. It calls the second method with `sendBuffer` as the third argument.

The second method can be called by the first, as mentioned above, or by method `requestViews` after we reconnect to the event source. The third argument in that call is msgBuffer. This method constructs an idGetView message in its third argument and sends it to the event server.

`boolean` **newFilter**`(String name, long filter)`

This method is called by the manager window thread when the user changes the filter for a view. This method constructs an idNewSelection message in `sendBuffer` and sends it to the event server.

`void` **deleteView**`(String name)`

This method is called by the manager window thread when the user closes a view. This method constructs an idCloseView message in `sendBuffer` and sends it to the event server.

`void` **closeSocket**`()`

If field `socket` is non-null, this method closes fields `socket`, `is` and `os`.

2.4.8 class EvtdspApp

`public class` **EvtdspApp** `extends Thread`

When this program is run as an application, this thread is invoked directly. When run as an applet, **EvtdspButton.actionPerformed**() starts this thread when the user clicks its button.

Data members:

```
boolean inAnApplet = true;
EvtdspButton parentButton = null; // Used when we are started
                        // by an EvtdspButton. Normally this field is null
                        // if and only if inAnApplet is false.
boolean userQuit = false; // Set true when the user wants
                        // to quit.
MgrWindow mgrWindow;     // The manager window.
EventThread eventThread;// The thread that will connect to the
                        // event server and receive events.
ErrWindow errWindow;    // A window where user errors are displayed.
EvtGlobal global;
StartupViewList startupList; // The list of views
                        // specified in the html page parameters.
CmdParams cmdParams;     // Used to process parameters.
                        // Allocated in the constructors.

final int defaultPort = 6066;

// host is set by EvtdspButton when we start as an applet,
// by readParams when started as an application.
String host;
String portString;

// Stuff to deal with platform-dependent bugs.
final int unknown = 0;
final int SunOS = 1;
final int Solaris = 2;
final int Windows95 = 3;
final int Window_16bit = 4;
final int FreeBSD = 5;
```

```
int os; // Set by the constructor to one of the constants above.
```

```
public EvtdspApp(String str, String[] args)
public EvtdspApp(String str, EvtdspButton parentButton)
```

The first constructor is used when the program is started as an application. The second when the program runs as an applet. Both constructors allocate objects **global**, **startupList** and **cmdParams**. They also determine the current operating system and set **os**. The only current use of **os** is in View.handleEvent.

```
public void run()
```

Algorithm of this method:

```
Process parameters.
if(running as an application and -SERVER not specified)
   Write an error message and exit.
Convert font size strings to integers.
Convert default color strings into Color objects.
Create a manager window object.
Create an EventThread object for communications
   with the event server process.
Start that thread.

Start and display any startup windows.
if(there are no startup windows)
   Make the manager window visible.
endif

repeat
   suspend this thread.
until the user quits

Destroy sockets, streams and windows.
```

The `suspend` in the algorithm above is broken when methods in the manager window object find that the user wants to quit and call this class's `wake` method. EvtdspButton calls `wake` when the user kills the program via that button.

```
public static void main(String[] args)
```

This method is called only when the program is run as an application. It creates an instance of this class, sets its `inAnApplet` to false, and starts it.

This method then waits for the thread to exit and calls `System.exit(0)`.

```
void wake()
```

This method sets `userQuit` to true and calls resume() to bring method `run` out of its `suspend` call. It provides a way for MgrWindow and EvtdspButton to notify this thread when the user wants to quit.

By setting `userQuit` to true, this method also signals the event thread to exit. That thread frequently checks the status of `userQuit`, and exits when it becomes true.

```
public void destroy()
```

This method closes any open sockets, streams and windows. It is called by this thread's **run** when the user quits.

`void` **fatalError**`(String msg)`

If this program is being run as an application, `msg` is written to stderr and the application exits.

If the program is run as an applet, `msg` is written to the text area on the html page. The applet returns after calling this function.

`void` **nonFatalError**`(String msg)`

This method displays `msg` in an ErrWindow. It is used to display error text to the user, such as

```
You must select a view before clicking on "Filter selected view".
```

If `errWindow` has not been allocated yet, this method creates it.

`errWindow` is placed just below the top of the manager window and made visible.

`protected void` **readParams**`()`

This method is called by **run** and it looks for values for its parameters. It uses cmdParams.getValue to obtain parameter values. That method examines the html page if this program is running as an applet and examines the command line otherwise.

`private void` **doViewSpec**`(String str)`

"str" specifies a startup view as described in Html parameters. This method identifies the fields in str and adds a view to the list maintained by class StartupViewList.

This method is called by readParams.

Methods
private int **findStart**(String str, int i)
private int **findEnd**(String str, int i)
Are used by readParams to find the beginning and end of substrings in its argument.

2.4.9 class EvtdspButton

`public class` **EvtdspButton** `extends Applet implements ActionListener`

This class is named as the CODE class in the html file. It creates buttons users can click to start and stop event display. It also creates a text area where status and error messages are displayed.

Data members:

```
Button startButton;         // Button for starting the event display.
Button quitButton;          // Button for killing the event display.
TextArea textArea;          // Shows status and error information.
final String appMainClass = "EvtdspApp"; // Name of the application.
EvtdspApp evtThread;        // The thread that will run the application.
```

`public void` **init**`()`

This method allocates objects **startButton**, **quitButton** and **textArea**. QuitButton is disabled at this time.

Method `actionPerformed` in this class is registered as the Action Listener for the two buttons.

`public void` **start**`()`

This method insures that the main application is accessible and is a Thread.

`public void` **actionPerformed**`(ActionEvent evt)`

This method implements the ActionListener interface. It is called when either the "Start" or "Quit" button is clicked.

When the "Start" button is clicked:

1. The "Start" button is disabled.
2. An EvtdspApp object is created and started.
3. The "Quit" button is enabled.

When the "Quit" button is clicked:

1. The "Quit" button is disabled.
2. Method `killTheApp` is called to kill the EvtdspApp thread.
3. The "Start" button is enabled.

`void` **eventExiting**`()`

This method is called by evtThread before it exits. This method disables the "Quit" button and enables "Start".

`protected void` **killTheApp**`()`

If evtThread is alive, this method calls its `wake` method and then waits for it to exit.

`public void` **destroy**`()`

```
This method overrides Applet.destroy. It kills evtThread
if it is alive.
```

`void` **appendMessage**`(String msg)`

This function is called by methods in EvtdspApp when they want to append to the text in **textArea**.

2.4.10 class EvtGlobal

This class does not define any methods. It defines values which are used by methods in many classes.

```
String textFont = "7X14";       // Default text font. May be reset
                                  // by the TEXT_FONT parameter.
String textFontSizeStr = "14";  // Default text font size. May be
                                  // Reset by the TEXT_FONT_SIZE param.
int textFontSize;               // Int version of textFontSizeStr.

String buttonFont = "Helvetica";// Default button font. May be reset
                                  // by the BUTTON_FONT parameter.
String buttonFontSizeStr = "14";// Default button font size. May be
                                  // reset by BUTTON_FONT_SIZE.
int buttonFontSize;             // Int version of buttonFontSizeStr.

// These colors can be reset by parameters VIEW_FG and VIEW_BG.
```

```
String defViewFGStr = "white";     // White foreground.
String defViewBGStr = "0, 0, 140"; // Dark blue background.

Color defViewFG = null; // Colors corresponding to defViewFGStr
Color defViewBG = null; // and defViewBGStr.
```

2.4.11 class FilterList

Data members:

```
FilterView first = null; // Pointer to the first view.
EvtdspApp app;           // This just gets passed to FilterView.
```

There is one object of this class, created by the constructor of the manager window. Its `first` field points to a linked list of FilterView objects.

A FilterView object is used to modify a view's event filter. They are created only when needed, and can be reused for different views. If the user does not modify any filters, the list remains empty. If she never brings up more than one FilterView at a time, only one Filterview is created.

A field in each FilterView tells which view's filter is being displayed.

```
void showFilter(View view)
```

If there is currently a FilterView for `view`, this method pops it to the top. Otherwise, it creates a FilterView for `view`.

Algorithm:

```
if(there's already a FilterView for view)
   Call FilterView.toFront()
   return
endif

if(all FilterViews are in use)
   Create a new FilterView and add it to the list.
endif
Call FilterView.show()
```

2.4.12 class FilterView

```
class FilterView extends Frame implements ActionListener
```

An object of this class is a window which allows the user to change the event filter of a view. These objects are created by the `showFilter` method of class FilterList, which manages a list of these objects linked by their `next` fields.

Data members:

```
static final int numEventTypes = 29; // Number of event types.
final int eventRows = 8;// Number of rows of event buttons.
View view = null;       // The View whose filter is displayed.
                        // null if this FilterView is unused.
FilterView next = null; // Link to the next FilterView.
EvtdspApp app;          // Used in the action method to
```

```
                          // invoke EventThread.newFilter
                          // if the user clicks "OK".
Checkbox[] eventBox;      // numEventTypes Checkboxes are allocated for
                          // this array. evnetBox[0] is used for event 1.

// Buttons for selecting groups of events.
Button allButton;
Button noneButton;
Button telemetryButton;
Button commandButton;
Button stolButton;

// Action buttons.
Button okButton;
Button resetButton;
Button cancelButton;
```

The window contains three panels. The top panel contains `numEventTypes` Checkboxes, one for each event type. The middle panel contains the group selection buttons, and the bottom panel contains the action buttons.

FilterView`(EvtdspApp app)`

This constructor creates the panels and buttons described above and packs them into the frame. It sets the font to EvtGlobal.buttonFont, and sets `this.app` to the argument `app`.

`void` **show**`(View view)`

This method calls **setChecks**, described below, to display `view`'s filter in the checkboxes, positions the window over `view` and makes the window visible.

`public boolean` **action**`(Event evt, Object arg)`

This method is called when the user clicks any button in the window. It responds as follows:

```
if("All" was clicked)
   All Checkboxes in eventBox are set to true.
else if("None" was clicked)
   All Checkboxes in eventBox are set to false.
else if("Telemetry" was clicked)
   The telemetry elements of eventBox, 0 through 6, are set to
   true. The others to false.
else if("Command" was clicked)
   The command elements of eventBox, 7 through 13, are set to
   true. The others to false.
else if("Stol" was clicked)
   The stol elements of eventBox, 14 through 18, are set to
   true. The others to false.
else if("OK" was clicked)
   The window is hidden.
   view.filter is set to the current state of the Checkboxes.
   if(new filter != the old filter)
```

```
         The new filter is passed to
         app.eventThread.newFilter.
      endif
      view is set to null to show that this object is not in use.
   else if("Reset" was clicked)
      The Checkboxes are set to display view.filter.
   else if("Cancel" was clicked)
      The window is hidden.
      view is set to null to show that this object is not in use.
   endif
```

`protected void` **setChecks**`()`

This method sets Checkboxes in the array `eventBox` to display `view`'s current filter.

`protected void` **saveChecks**`()`

This method sets `view`'s filter to reflect the current state of Checkboxes in the array `eventBox`.

2.4.13 class MgrWindow

`class` **MgrWindow** `extends Frame implements ActionListener`

This class implements a window similar to the manager_shell widget in dsp_evtdsp. It lists the current views and provides "Dismiss" and "Quit" buttons. The menu of dsp_evtdsp is reduced to three buttons here because of the reduced functionality of this program. The three buttons are:

- New view...
- Filter selected view
- Delete selected view

The "Dismiss", "Filter selected view" and "Delete selected view" buttons are inactive when there are no views.

Data members

```
EvtdspApp app; // The application to notify when the user wants
               // to quit.
// The five buttons.
Button newViewButton;
Button filterViewButton;
Button deleteViewButton;
Button dismissButton;
Button quitButton;

List viewNameList;      // Java component displaying the list of
                        // current views.
ViewList viewList;     // A linked list of View objects.
ViewDialog viewDlg;     // Dialog for querying the user for
                        // a new view name.
ViewPopup viewPopup;    // The popup menu used by all views.
FilterList filterList; // List of windows allocated for
```

```
                              // specifying filters.
    // Arguments to makeNewView.
    static final boolean showError = true;
    static final boolean noError = false;
```

`public` **MgrWindow**`(EvtdspApp app)`

Actions of this constructor:

1. The argument is used to initialize **app**.
2. All visible components of the manager window are created and packed into the frame.
3. The "Dismiss", "Filter selected view" and "Delete selected view" buttons are disabled.
4. viewList, viewPopup and filterList are created.
5. An object of the inner class `DWAdapter` is created and registered as a window listener for this frame.

`void` **makeNewView**`(String viewName, boolean printError)`

This method is called to create a new view named `viewName`. If that view already exists, no view is created. If `printError` is true in this case, an error is displayed to the user.

This method is called from:

1. EvtdspApp.run() for each startup view specified in the html file. `printError` is false in this case, so no error message is displayed for duplicate views.
2. ViewDialog.action() when the user enters a view name in that dialog. `printError` is true in this case, so an error message is displayed.

Before creating the new view, this method searches for the view in the StartupViewList. If found, its entry in that list is passed to the View constructor. This insures that the view will get the properties specified in the html file.

After creating the view, the method does the following:

1. Make the view visible.
2. Add the view to the list of current views.
3. Enable all manager window buttons. This is necessary only when there were previously no views.
4. Add the view to the text displayed in the manager's window.

`void` **deleteView**`(View view, int index)`

This method deletes `view`. If `index` is >= zero, it is the view's index in `viewNameList`. If `index` is less than zero, this method searches for `view` in `viewNameList`.

This method is called when the user clicks the manager window's "Delete selected view" button or "Close" in the view's frame menu.

The following actions are carried out:

1. Hide `view`.
2. Call `app.eventThread.deleteView` to tell the event server to stop sending events to this view.
3. If `index` is less then zero, find `view` in `viewNameList`.

4. Remove `view` from `viewNameList`, the text shown in the manager window.
5. Remove `view` from `viewList`, the linked list of View objects.
6. Call `view.dispose` to free resources.
7. If there are now no views, disable the "Dismiss", "Filter selected view" and "Delete selected view" buttons.

`void` **eventMsg**`(MsgBuffer buf)`

This method is called by the eventThread when it gets an event from the event server. `buf` is the buffer that received the event message. The next string to be extracted from `buf` is the text of the event message. Following strings are the names of views where that text should be displayed.

If the event text begins with two decimal digits, this method assumes they give the message type and removes them. If the event text ends with a linefeed, this method removes that.

The resulting text is appended to all views named by calling their `textArea.appendText` methods.

`public void` **actionPerformed**`(ActionEvent evt)`

This method implements the ActionListener interface. It responds to clicks on buttons in the manager window.

`void` **closeAllViews**`()`

This method is called by `EvtdspApp.destroy` when EvtdspApp exits. It calls `dispose` for all currently open views and for viewDlg if it exists.

`void` **destroyAllViews**`()`

This method is called by EvtdspApp.run just before it exits. This method disposes of all views.

2.4.14 class StartupView

An object of this class contains all information about a startup view. A startup view is one specified in the html file in a VIEWn parameter. Class StartupViewList manages the linked list of these objects. (See Html support for EvtdspApp for more information on VIEW parameters.)

Data members:

```
String name;        // The view's name.
long filter;        // The view's event filter.
int cols;           // Number of character columns in the window.
int rows;           // Number of text lines in the window.
int maxRows;        // Number of lines to store. Old lines are
                    // discarded when more than this arrive.
String FGColorStr;  // Foreground color.
String BGColorStr;  // Background color.
boolean show;       // Make this view visible at startup?
// The next two fields specify the top left corner of the view.
// They are ignored if either is negative.
```

```
int top;
int left;
StartupView next;   // Link. Managed by class StartupViewList.
```

There are no methods in this class.

2.4.15 class StartupViewList

This class manages a list of StartupView objects containing all startup views specified in the html file in VIEWn parameters. (See Html support for EvtdspApp for more information on VIEW parameters.)

This list is used:

1. At startup. All startup views whose `show` fields are true are displayed.
2. The list is searched every time the user creates a new view. If the new view's name is found in the list, the values stored there are used for the view.

Data members:

```
StartupView first = null; // Pointer to the first view.
EvtGlobal global;          // Pointer to the instance of EvtGlobal
                            // created by EvtdspApp.
```

The StartupView objects are linked by their `next` fields.

```
void add(String name)
```

```
void add(String name, String evtSpec)
```

```
void add(String name, String evtSpec, int cols, int rows)
```

```
void add(String name, String evtSpec, int cols, int rows,
      int maxRows)
```

```
void add(String name, String evtSpec, int cols, int rows,
      int maxRows, String fg, String bg, boolean show)
```

```
void add(String name, String evtSpec, int cols, int rows,
      int maxRows, String fg, String bg, boolean show,
      int left, int top)
```

The first five versions of this method add default values and call the sixth version.

If there is already a view in the list named `name`, nothing is done. Otherwise a new StartupView object is created, its fields are filled from the arguments to this method, and it is linked into the list.

```
private long getFilter(String str)
```

If `str` is a valid event specification, this method returns the filter specified. Otherwise it returns defFilter. A valid specification is a list containing one or more of:

- +* => Add all events.
- -* => Remove all events.
- +N => Add event number N.
- -N => Remove event number N.

This method is called by this class's `add` method.

`StartupView` **nextStartup**`(StartupView prev)`

If `prev` is null, this method returns the first StartupView in the list. Otherwise, it returns the StartupView after "prev". If "prev" is the last view in the list, null is returned.

2.4.16 class TextQueue

An object of this class maintains the list of event messages which can currently be viewed in one CanvasTextArea object. Each event message is stored in an EventMsg object. Messages can be added to this queue, but they are deleted only when there's a queue overflow. As a result, there's no need for a "head" pointer in the queue.

Data members:

```
EventMsg[] line; // The array of viewable event messages.
int maxLines;     // Number of lines allocated for the array line[].
int tail;         // Index in line where the next data will be put.
boolean full;     // True when there is data in all lines.
int blockNum;      // Sequence number of the most recently created
                  // block of lines. Used to tell when two threads are
                  // requesting blocks simultaneously.
```

TextQueue`(int maxLines)`

This constructor assigns its argument to data member `maxLines`. It then allocates an array of EventMsg objects and assigns it to `line`. `tail` is set to 0 and `full` to false.

`boolean` **append**`(String str)`

This method is called by the CanvasTextArea method `appendText`. It puts `str` at the end of the queue in an EventMsg object.

`str` may start with one or more of the following prefixes:

1. A BEL character. These are removed.
2. "<ESC>[FCm" where:
   - C = a digit character specifying a color.
   - F = '3' if C specifies a foreground color, '4' if it specifies a background color.
   - The <ESC>, '[' and 'm' characters are constants.

This method reads through the prefixes and uses them to set the `fg` and `bg` fields of the EventMsg at the tail of the queue. It then removes the prefixes from `str` and puts it into the `text` field of the EventMsg.

Finally, the tail pointer is incremented and `full` may be set to true.

If a BEL character was found, true is returned. False otherwise.

`Color` **getColor**`(char c)`

The argument `c` is a digit character. This method returns the color it specifies. `c` values '0' or '7' specify defaults, so this method returns null. Null is also returned if `c` is not a a recognized color.

`int` **length**`()`

This method returns the number of strings in the queue. If the queue is full, this is `maxLines`. Otherwise it is `tail`.

```
BlockOfLines requestBlock(int newest, int oldest)
```

CanvasTextArea.paintText wants to print the lines from "newest" to "oldest". Those are non-negative integers giving offsets from the most recently received line. This method initializes a BlockOfLines object and returns it. If there is a problem with the arguments, null is returned. Null is also returned if none of the requested lines is in the queue.

```
EventMsg nextLine(BlockOfLines blk)
```

This method returns the next line of `blk` or null. If a line is returned, `blk` is shortened so it no longer contains the returned line. Usually this involves returning `line[blk.newOffset]` and decrementing `blk.newOffset`, mod maxLines.

However, there are many variations:

If the block contains the tail pointer, some of the lines in the originally-requested block are no longer in the queue. Therefore the block is shortened by setting `blk.oldOffset` equal to `tail`.

It is possible that none of the remaining lines of the block are still in the queue. Null is returned in this case.

If the line being returned is the last line in the block, both `blk.newOffset` and `blk.oldOffset` are set to -1, so the next call to this method will return null.

If `blk.blockNum` is less than `blockNum` in this object, a block was requested by some other thread after the current block. This method returns null to stop painting from this block.

This method assumes that views are drawn from the bottom up, that is, starting with the most recent of the visible lines.

2.4.17 class View

```
class View extends Frame
```

This class contains information on one view.

Data members:

```
String name;   // The view's name.
long filter;   // Filter specifying which events are
               // displayed in the view.
CanvasTextArea textArea; // The visible view.
MgrWindow mgr; // Pointer to the manager window. This
               // pointer is the same in all views.
boolean popOnBeep = true; // When true, this view is
               // popped to the top when it displays a
               // message containing a BEL character.
```

There are two constructors:

```
View(String name, MgrWindow mgr)
View(StartupView st, MgrWindow mgr)
```

The first constructor creates a StartupView object using default values for properties. Both constructors call `private void` **construct**`(StartupView st, MgrWindow mgr)`

which sets the `name`, `filter` and `mgr` fields and passes properties from `st` to the constructor for `textArea`.

`void` **appendText**`(String str)`

This method is called by MgrWindow.`eventMsg` when a new event message is received from the event server. `str` is the text of the new message. This method adds it to this view.

If `str` contains a BEL character and popOnBeep is true, this frame is popped to the top.

`str` is passed to text.append so it is appended to the text queue. Subsequent actions:

`void` **toggleBeepPop**`()`

Toggles the value of `popOnBeep`.

`public boolean` **handleEvent**`(Event evt)`

This method handles the following events:

1. If the user clicks on "Close" in `textArea`'s frame menu, a WINDOW_DESTROY event is received and this method calls `mgr.deleteView`.
2. If the user resizes `textArea`, a WINDOW_MOVED event is received and this method calls `textArea.sizeChanged`.
3. If the user moves the mouse out of the window on a Windows 95 system, this method calls `textArea.sizeChanged` and `repaint` on the text area. This is necessary because of a bug in the Windows 96 java implementation. Frames do not get WINDOW_MOVED events, so we cannot detect window resizing. As a workaround, we check the size of the window every time the cursor leaves the window.

2.4.18 class ViewDialog

`class` **ViewDialog** `extends Frame extends ActionListener`

One object of this class is created by the manager window. It is used to query the user for the name of a new view and contains the following data members:

```
TextField viewName;  // Field where user enters text.
Button okButton;
Button cancelButton;
MgrWindow mgr;       // Pointer to the manager window. Used to call
                     // the manager's makeNewView method when
                     // the user clicks "OK".
```

This class could be derived from Dialog, but when run as an applet, the first showing of a Dialog redraws every window on the screen (At least under Solaris.)

ViewDialog`(Frame parent)`

This constructor sets `mgr` to the argument and creates the buttons and text area.

`public void` **actionPerformed**`(ActionEvent evt)`

This method handles clicks on the "Cancel" and "OK" buttons and recurn in the text field.

2.4.19 class ViewList

This class implements a linked list of all current Views. Each object in the list is an instance of class ListElement. Those objects have two data members:

```
View view;         // Pointer to a view.
ListElement next; // Link to the next list element.
```

Class ViewList has only one data member:

```
ListElement first;  // Pointer to the first view.
```

Only one object of ViewList is instantiated, by the constructor of class MgrWindow.

All of the ViewList methods are synchronized because two threads could access the list simultaneously. List modifications are made in response to user actions in the manager window. (I guess that means they're done by the Java AWT thread.) The eventThread needs a list of current views when it reconnects to the event server after a network problem. It gets that list by repeatedly calling the `viewNumber` method.

```
synchronized void add(View view)
```

If `view` is not currently in the list, this method adds it to the end of the list.

```
synchronized void delete(View view)
```

If `view` is currently in the list, this method deletes it.

```
synchronized View getView(String name)
```

If there is currently a view in the list named `name`, this method returns a pointer to it. Otherwise it returns null.

```
synchronized View viewNumber(int pos)
```

This method returns the view at postion `pos` in the list. The first view is at position zero. If there is no view at position `pos`, this null is returned.

When the event thread reconnects to the event server after a break, its `requestViews` method needs a list of the current views. It gets the list by calling this method repeatedly with increasing values of `pos` until null is returned. If the user were to delete a view while these calls were being made, a different view could be left out of the list requested from the event server. At the moment, this does not seem worth fixing because it is both unlikely and its consequences are not severe. The user would have to request the view again.

```
synchronized View firstView()
```

This method returns the first view in the list, or null if the list is empty.

```
synchronized boolean isEmpty()
```

This method returns true if there are no views in the list.

2.4.20 class ViewPopup

```
class ViewPopup extends Frame implements ActionListener
```

This class implements the popup menu a user gets by clicking in an event view window. Since the popup menu is visible in only one view at a time, only one instance of this class is created, by mgrWindow. It is reused for all views.

Data members:

```
MgrWindow mgr;  // Pointer to the manager window. Needed if
                // the user selects "filter", "delete" or
                // "manage".
View view;      // The view this menu is currently attached to.
                // This member is set by CanvasTextArea.handleEvent
                // when this object is made visible.
// The five buttons of the menu:
Button autoButton;   // Toggle autoscroll.
Button filterButton; // Modify the view's filter.
Button beepButton;   // Modify view's reaction to BEL.
Button deleteButton; // Delete the view.
Button manageButton; // Bring up the manager window.
```

`ViewPopup(MgrWindow mgr)`

This constructor creates the five buttons of the menu and sets the font used. It also creates an instance of the inner class `DWAdapter` and registers it as a window listener.

`public void actionPerformed(ActionEvent evt)`

This method is called when the user clicks one of the menu's five buttons:

- If autoscroll, view.textArea.toggleAutoscroll() is called.
- If filter, mgr.filterList.showFilter(view) is called.
- If beep, view.toggleBeepPop() is called and the label on `beepButton` is changed to indicate what will happen when the user clicks it.
- If delete, mgr.deleteView(view, -1) is called.
- if manage, the manager window is placed over the click and mgr.show() is called.

`protected String beepText(boolean popOnBeep)`

This method sets the text of `beepButton` based on the value of `popOnBeep`. If it is true, the text is set to "Stop pop". Otherwise to "Pop on Beep".

`class DWAdapter extends WindowAdapter`

This inner class defines method:

`public void windowClosing(WindowEvent event)`

This method hides the popup menu.

2.5 Puzzles

Some oddities in the current code.

Resizing a view causes the following problems on SunOS java:

1. After the view is resized, the Component.location() method returns 0,0. If the view is moved, location again returns the location of view's the upper-left corner. The location method is used to position popup menus and the manager window under the cursor.
2. After the view is resized, it stops receiving GOT_FOCUS and LOST_FOCUS.

If an event message is written to a view at precisely the same time as the view is resized, the window's text is written twice. The second write is offset from the first and the screen is not cleared between the two writes, so nothing can be read.

The two writes are done by different threads. Debugging output shows that the two calls to the paint routine do not overlap. I assume that what is happening is that update is called by the two threads, both instances of update clear the screen and then both call paint.

Can anything be done about this?

Finally, a functional question. The current code of the EvtdspButton applet does not redefine start and stop methods. I did this because I think we want to give users the ability to iconize the browser or go to another page without losing the event view windows.

Opinions?

3 evtforward

Evtforward is a java background application that forwards event messages from a source to multiple consumers. Typical sources are dsp_evtdsp or another instance of evtforward. Typical consumers are dsp_evtdsp, EvtdspApp or another instance of evtforward.

Evtforward (or a chain of evtforwards) makes it possible for a process to get event messages from another process to which it cannot make a socket connection. For example, evtforward running on the gateway between sub-networks would allow machines on one subnet to get events from the main TCW on the other subnet.

Evtforward sends and receives all messages over tcp sockets using the same message format as dsp_evtdsp.

evtforward uses source files '`evtforward.java`' and '`MsgBuffer.java`'.

3.1 Evtforward command line arguments

One of these two arguments must be given:

```
-source <address> [<port number>]
      Specifies the address and optionally the port number where
      evtforward will get event messages. The default port number is
      6066, the default port used by dsp_evtdsp.

-protectedsource <address> <port>
      Is used instead of -source when this application cannot
      connect to the event source's socket because of a firewall.
      <address> is the source's address, it is used in messages.
      <port> is the port the event source will try to connect to.
      If the event source is another evtforward, it must be given
      a -getclient argument with this port number. (-getclient is
      described below.)
```

Other command line arguments:

```
-help
      Prints a message describing command line arguments to stdout
      and exits.

-localPort <port number>
      Specifies the server port evtforward will open for consumers.
      The default is 6067.

-getclient <address> <port>
      is used when an event client cannot connect to this
      application's server port because of a firewall. This
      application will connect to <address>, <port> and send events
      there.
      If that client is an evtforward, it must be started with
      -protectedsource using the same port number.
```

```
-wait
     By default, if evtforward cannot connect to the socket
     specified in -source, it exits immediately. When this argument
     is present, evtforward waits for a server to open the socket.

     If the socket is closed at a later time, evtforward waits for
     it to be reopened whether this argument is present or not.

-checkServer
     If evtforward cannot open its server port, it always exits.
     By default, it also writes an error message.
     When this flag is present and the port cannot be opened because
     it is already in use, no error message is written before exit.
     This may be useful in scripts when it is not known whether an
     instance is already running.

-debug
     Causes some debugging messages to be written to stdout.

-queueSize <value>
     Specifies the depth of the event queue, the default value is 1024.
```

Error messages are written to stderr.

3.2 Overall structure of evtforward

Main algorithm of evtforward (contained in the evtforward constructor):

```
Process command line arguments.
Exit if neither "-source" nor "-protectedsource" was not specified.
Allocate an event queue.
if(protectedsource)
   Open a socket the event source can connect to.
else
   Connect to the event source socket.
   If that connection fails and "-wait" was not specified
      Exit.
endif
Start a SourceThread to monitor that connection.
if(any -getclient arguments)
   Start Client threads to connect to those clients.
Open a server socket.
Repeat forever
   Wait for a client to connect to that server socket.
   Start a Client thread. It monitors messages from the client
      and starts a ClientEvtThread to send events to the
      client.
end repeat
```

Since there is no user interface to a running evtforward, it must be killed manually.

SourceThread is a class containing methods for communicating with the event source. Only one instance of this class is created, by the evtforward constructor.

On startup, SourceThread requests its only view from the source. The filter associated with that view changes as clients change their requests. That filter is always the union of the filters of all client views.

After startup, `SourceThread.run` simply waits for event messages from the source and puts them into the event queue.

If SourceThread finds that it has lost contact with the event source, it puts a warning message into the event queue which is displayed in all views of all clients. It then calls `reconnect`, whose actions depend on whether -source or -protectedsource was used at startup. If -protectedsource was used, `reconnect` simply waits for the event source to connect. If -source was used, `reconnect` periodically tries to reconnect to the source's server socket. The frequency of those reconnection attempts is determined by the class constant `waitMillis`, currently set to 5 seconds.

After initialization, the only message sent to the event source is a request to change the events this process receives. Those requests are sent by method `sendNewFilter`, defined in this class, but run in the client thread that changed the event filter. Since `sendNewFilter` is called asychronously by client threads, it uses a separate buffer for constructing the message to the source (to prevent corruption of the buffer used by the source thread) and is synchronized (to keep two clients from corrupting each other's messages).

Two threads are involved with each client. The first is an instance of class Client, started by `evtforward`'s constructor when it first learns about a client. That happens when a `-getclient` argument is processed or when a client connects to the server socket. The `Client` thread responds to messages from the client.

The `Client` thread starts a second thread, of class `ClientEvtThread`, to send event messages to the client.

`Client` receives three kinds of messages from the client:

1. GetView messages are received when the client starts a new view.
2. NewSelection messages are received when the client changes the set of events displayed in one of its views.
3. CloseView messages are received when the client closes a view.

`Client` maintains a list of the client's views and the event filter associated with each.

When the client exits, it simply closes its socket. There is no closing message to this process. Either of the two threads associated with the client may discover that the client has closed its socket. That thread calls `Client.shutdown` which removes this client from the client list (described below) and closes the socket and streams to the client.

After ClientEvtThread calls `shutdown`, it exits. After Client calls `shutdown`, its actions depend on whether we learned about the client from `-getclient` or from a connection to the server socket. In the latter case, Client exits. In the former, it calls `reconnect`, which repeatedly tries to reconnect to the client. After connection, a new `ClientEvtThread` is started for the client.

Clients specified in `-getclient` command line arguments are called "excluded" clients because a firewall prevents them from connecting in the normal manner, to the server socket.

Two fields in a `Client` object are used only by excluded clients: `addr` and `port` specify the server socket opened by an excluded client. A `Client` object repeatedly tries to connect to that socket any time it loses contact with an excluded client. Fields `addr` and `port` are not used by ordinary clients.

The ClientList class implements a linked list of clients currently connected to this process. Its methods are called by the client threads to add themselves to the list when they connect to a client and to delete themselves from the list when connection is lost.

This class maintains `filterUnion`, the union of all filters of all views of all clients. When `filterUnion` changes, a request is sent to the event source to change the filter of the one view this process has from the event source.

Since multiple threads modify and access this class's list, all methods are synchronized.

`evtforward`'s constructor allocates an event queue, an object of class EvtQ. When the source thread receives an event message from the event source, it puts the message into this queue. That wakes all ClientEvtThread objects which were waiting for the next event message. Each ClientEvtThread examines the id of the new message and the filters of its client's views and decides whether to forward the event to its client.

The size of the queue is fixed by an argument to its constructor. Messages are removed from the queue only when it overflows, at which time the oldest message is discarded.

`sequenceNum` is a data member of `EvtQ` which is incremented each time a message is added to the queue. Each ClientEvtThread object knows the sequence number of the last message it examined, and requests messages from the queue by sequence number. When the queue returns a message to a ClientEvtThread, it includes the message's sequence number.

This scheme allows a client to temporarily get behind in processing messages and still not lose any. When a client requests a sequence number higher than `sequenceNum`, it blocks in `EvtQ.get` until that message arrives.

There is a reserved sequence number that a ClientEvtThread uses in its first request meaning "Give me the most recent message and tell me what its number is.".

This queue (and the ClientEvtThreads) exist to protect the source thread from slow clients. If the source thread wrote to client sockets and a client got so far behind that its socket filled up, the source thread would hang. With the current method, only that client's ClientEvtThread hangs.

3.3 Class evtforward

class evtforward data members:

```
String sourceAddr;     // Address of the event source to connect to.
int sourcePort = 6066; // Default port used for connecting to the source.
int localPort = 6067;  // Default number of this server's port number.
boolean debug = false; // When true, debug messages are printed.
boolean waitForSource = false; // Set by -wait. see command line args.
boolean checkServer = false;   // Set by -checkServer. See command line args.
boolean protectedSource = false; // Set by -protectedsource.
ServerSocket dataSocket;       // Used only when -protectedsource is.
                                // The socket where we wait for the data source.
```

```
ExcludedClient exClient;       // The list of clients specified in
                                 // -getclient command line arguments.
ClientList clientList;        // List of connected clients.
EvtQ evtQ = null;                // Queue of the most recent events.
SourceThread sourceThread;      // Thread that receives event messages.
String noDataWarning             // Text of the message sent to all views when
                                 // we lose contact with the event source.
final int waitMillis = 5000;   // When trying to reconnect with a server socket,
                                 // we sleep this many millisec between attempts.
protected int queueSize;   // the size of the EvtQ, defaults to 1024
String version;                // Version number displayed at startup.
```

class evtforward methods:

public static void **main**(String[] args)

Creates an instance of evtforward, then exits when the constructor returns.

evtforward(String[] args)

The whole application runs in this constructor.

Algorithm:

```
Process command line arguments.
Exit if neither "-source" nor "-protectedsource" was not specified.
Allocate an event queue.
if(protectedsource)
   Open a socket the event source can connect to.
else
   Connect to the event source socket.
   If that connection fails and "-wait" was not specified
      Exit.
endif
Start a SourceThread to monitor that connection.
if(any -getclient arguments)
   Start client threads to connect to those clients.
Open a server socket.
Repeat forever
   Wait for a client to connect to that server socket.
   Start a Client thread. It monitors messages from the client
      and starts a ClientEvt thread to send events to the
      client.
end repeat
```

static void **processArgs**(String[] args)

Processes command line arguments. This method should set evtforward's `SourceAddr` field and may set the `SourcePort`, `localPort`, `protectedSource`, `waitForSource` and `debug` fields. It may add elements to `exClient`, the list of clients specified with `-getclient`. If "-help" is found, exit is called after the help text is printed.

void **addExcludedClient**(String addr, int port)

Adds `addr`, `port` to `exClient`, the list of excluded clients.

static void **printHelp**()

This method is called by processArgs when the -help command line argument is found.

3.4 Class ClientEvtThread

This class extends `Thread` and is an inner class of Client. When we establish contact with a client, the Client thread for the client creates an instance of this class. An object of this class waits for messages to be put into the event queue and then sends the interesting ones to the client.

Data members:

```
Client client;       // The Client thread paired with this thread.
MsgBuffer evtBuffer; // Event messages to the client are
                      // constructed here.
```

Methods:

ClientEvtThread(Client clientThread)

This constructor sets `client` to the argument and allocates `evtBuffer`.

public void **run**()

This method allocates an EvtQRequest object for use in requesting messages from the event queue. It sets the request's sequence number to 0, meaning the first request is for the most recent message.

It then repeatedly requests a message.

If the message is in the queue, it is put into the request structure along with its sequence number. The event text and id are passed to `checkEvent` which will decide whether the client wants the message and it so, sends it. The sequence number is incremented for the next request.

If the message is not in the queue, the request (a call to `EvtQ.get`) blocks until the next message arrives.

Every time this method sends a message to the client and every time its `EvtQ.get` call returns, it checks whether `client.clientEvt` has changed. That means that the connection to the client has broken and it is time for this thread to exit.

void **checkEvent**(String eventText, long eventBit)

This method is called by `run` after it gets an event message from the queue. One bit in `eventBit` is set to indicate the event's type. This method calls `getViews` for a list of this client's views which are interested in the event. If any are interested, the list is passed to `sendEvent`.

void **sendEvent**(String eventText, String viewList)

This method constructs an event message in `evtBuffer` with `eventText` in the text field and `viewList` as the list of views. It sends the message over `os` to the client.

If an attempt to send the message to the client fails because the client has closed the socket, this method calls `client.shutdown` to close our end of the connection.

3.5 Class ClientList

Class ClientList maintains a linked list of OneClient objects. There is an object in the list for each currently connected client. ClientList is an inner class of evtforward.

Class ClientList data fields:

```
protected OneClient firstClient; // The first client in the list.
long filterUnion = 0L;           // Union of the filters of all views. This
                                   // is passed to SourceThread's sendNewFilter
                                   // method every time it changes.
```

This list is accessed by all client threads, so all methods are synchronized. They are:

synchronized void **addClient**(Client thread)

This method adds `thread` to the list of clients. Current code assumes that `thread` is not already in the list.

synchronized void **deleteClient**(Client client)

This method removes `client` from the list if it is there. If a client is removed, `checkFilterUnion` is called because the removal may have changed `filterUnion`.

synchronized protected long **getFilterUnion**()

This method returns the union of all filters of all views.

synchronized void **checkFilterUnion**()

If the current union of all view filters is different from `filterUnion`, this method resets filterUnion and passes it to the source thread's `sendNewFilter` method so the event source is notified.

3.6 Class Client

Class `Client` extends `Thread` and is an inner class of `evtforward`. One Client object is created for each client.

There are two kinds of client. The first kind of client makes itself known by connecting to evtforward's server socket. A `Client` object is created when this kind of client connects. When the client breaks the socket connection, the `Client` object's thread also exits and evtforward has no memory of the client.

The second kind of client is called an "excluded" client. It is specified in a `-getclient` command line argument because it is not able to connect to evtforward's server socket. During initialization, a `Client` object is created for every excluded client. These objects never exit. When they are not in contact with the client they continually try to connect to a server socket opened by the client. More details are given below.

The `run` method starts a ClientEvtThread and then simply waits for messages from the client. The ClientEvtThread waits for events to be added to the event queue and sends them to the client.

Four kinds of input can be received from the client:

1. A request for a new view.
2. A request to close a view.

3. A request to change the event filter of a view.
4. A notice that the client has closed the socket. This is sent by the MsgBuffer class.

class Client data members:

```
protected Socket socket;          // Socket to the client.
protected DataInputStream is;     // Input stream from the client.
DataOutputStream os;              // Output stream to the client. This is
                                  // used by the ClientEvtThread.
String name = null;               // This client's name.
String ownName = null;            // The name the client knows this process by.
MsgBuffer msgBuffer;              // All messages from the client go here.
ClientView firstView = null;      // Linked list of views.
ClientEvtThread clientEvt;        // The thread that sends events to this
                                  // client.
// The next two fields are used only by excluded clients. They specify
// the server socket opened by the client, which this thread connects to.
protected String addr = null;
protected int port = -1;
```

class Client methods:

Client(Socket sock)

This constructor is used when a normal client connects to our server socket. It calls `openStreams` and `commonInit`.

Client(String addr, int port)

This constructor is called to created an excluded client when a `-getclient` command line argument is processed. It sets `addr` and `port` and calls `commonInit`.

protected void **commonInit**()

This method is called by both constructors. It allocates `msgBuffer` and starts the thread for this object.

public void **run**()

Algorithm for this method:

```
if(this is an excluded client)
   Wait until we can connect to its server socket.
   Call openStreams().
endif
Create and start a ClientEvtThread for this client.
Repeat forever
   Wait for a message from the client.
   if(the client has exited)
      Call shutdown to clean up.
      if(this is an excluded client)
         Wait until we can connect to its server socket.
         Call openStreams().
      else
         Return. This terminates the thread.
      endif
      else
         Extract these fields from the message: type, sending display,
            receiving display, and view name.
         if(this is the first message from the client)
            Set the name and ownName fields of this Client.
         endif
         if(the message is GetView or NewSelection)
            Extract the filter field from the message.
            Call method addView to modify the view list.
         else if(the message is CloseView)
            Call method deleteView to modify the view list.
         endif
         Call ClientList.checkFilterUnion because this message may
         have changed the filter union.
      endif
   end repeat
```

protected void **openStreams**()

Opens input and output data streams for `socket`.

synchronized void **shutdown**()

This method is called when the client closes its end of the socket. The method calls `closeSocket` to close this end of the socket and the data streams. It calls ClientList.deleteClient to remove this client from the list. If debugging is enabled, a message is written to say this client has left.

This method can be called by either this clientThread or by its associated ClientEvtThread. Therefore it is synchronized.

void **closeSocket**()

This method is called by `shutdown`, and as described with that method, it can run in either the Client or ClientEvtThread.

If `socket` is non-null on entry, this method closes it and streams `is` and `os`.

If this method is called by the `Client` thread, there must be some mechanism for telling the `ClientEvtThread` to exit. The mechanism used here is to set `clientEvt` to null. `ClientEvtThread` continually compares that variable to its active thread. If they differ, `ClientEvtThread` exits.

synchronized void **addView**(String viewName, long filter)

If there is not a view named `viewName`, this method creates a ClientView structure, sets all fields, and adds it to this client's list. If there is already a view named `viewName`, this method just changes its filter.

If this process is not connected to the event source at the moment, the string `noDataWarning` is sent to the view.

This method is synchronized because method `getViews`, which is called by the ClientEvtThread associated with this client, traverses the view list which this method modifies.

synchronized void **deleteView**(String viewName)

This method deletes `viewName` from the list of views.

This method is synchronized because method `getViews`, which is called by the ClientEvtThread associated with this client, traverses the view list which this method modifies.

synchronized String **getViews**(long eventBit)

This method returns a string containing the names of views of this client which get the event specified in `eventBit`. Each name in the string is null-terminated. This method is called by `ClientEvtThread.checkEvent` when a message is taken from the event queue.

If `eventBit` is zero, all views are put into the string.

This method is synchronized so it does not run concurrently with `addView` or `deleteView`.

protected Socket **connectToClient**()

Attempts to connect to an excluded client's server socket. If the attempt fails because no process is listening on `addr, port`, this method sleeps for `waitMillis` milliseconds and tries again. Those steps are repeated until a successful connection is made, in which case the socket is returned, or until some error occurs, in which case null is returned.

protected boolean **reconnect**()

This method is called after we lose contact with an excluded client. It does the following:

1. Calls `connectToClient` to reestablist the socket connection.
2. Calls `openStreams` to create input and output data streams.
3. Starts a ClientEvtThread to send event messages to the client.
4. Calls ClientList.`addClient` to add this client to the list of clients.

3.7 Class ClientView

This class is an inner class of evtforward.

An object of class ClientView describes one view of a client. The `next` field allows linked lists to be formed, and every Client object contains a pointer to one linked list of ClientView objects.

class ClientView data members:

```
String name;       // The view's name.
long filter;      // The view's event filter.
ClientView next;  // Link to the next view of the same client.
```

This class defines only one method, a constructor, which sets the three data fields.

3.8 Class EvtQ

One object of this class exists, created by the evtforward constructor. It implements a queue of the most recent event messages received from the event source. The source thread puts messages into the queue and ClientEvtThread objects read them.

The queue is fixed-length, with the length specified in the constructor. A message is not removed from the queue until it is the oldest message in the queue and there is a queue overflow.

Class EvtQ data members:

```
EvtQElement[] q; // The array that implements the queue.
int maxQ;         // The number of elements allocated for "q".
int tail;        // Index in q where the next data will be put.
boolean full;    // True when q is full.
long sequenceNum;// The sequence number of the most recently
                 // added message.
```

Class EvtQ methods:

EvtQ(int maxQ)

This constructor allocates an array of `maxQ` EvtQElement objects. It also initializes `tail` to 0, `full` to false and `sequenceNum` to -1.

synchronized void **put**(String eventText, long eventBit)

This method puts its arguments into the array element pointed to by `tail`, increments `tail`, `sequenceNum` and may set `full` to true.

Finally, it calls `notify` to wake any ClientEvtThreads which are waiting for this event message.

synchronized void **get**(EvtQRequest req)

This method tries to get the message with sequence number `req.sequenceNum` from the queue and put it into `req.el`.

If `req.sequenceNum` has not been received yet, the caller blocks until it is received. In all other cases, some message is put into `req` and `req.sequenceNum` is set to the number of the message.

if `req.sequenceNum` is no longer available, the oldest message which is still available is put into `req`.

3.9 Class EvtQElement

An object of class EvtQ contains an array of objects of this class.

Data Members:

```
long eventBit;    // One bit in this field is set to specify the
                  // event's id.
String eventText; // The text of the event.
```

No methods are defined for this class.

3.10 Class EvtQRequest

ClientEvtThread objects pass a pointer to one of these objects to EvtQ.`get` when they request an event message from the event queue.

Data members:

```
long sequenceNum; // Sequence number of the event requested.
EvtQElement el;   // An element from the event queue is put here
                  // by EvtQ.get.
```

When `EvtQ.get` returns true it always puts data into the `el` field and changes the `sequenceNum` field if the message put into `el` is different from the one requested.

No methods are defined for this class.

3.11 Class OneClient

An object of class OneClient is one element in the list of clients maintained by ClientList. This class is an inner class of `ClientList`.

class OneClient data members:

```
Client client;        // Pointer to the client.
OneClient next;             // Pointer to the next object in the list.
```

class OneClient does not define any methods.

3.12 Class SourceThread

One SourceThread object is allocated. It handles all communication with the event source process. This class extends `Thread` and is an inner class of `evtforward`.

SourceThread data members:

```
static String sourceAddr; // Address used when connecting to the
                          // the event source.
int sourcePort;          // Port used when connecting to the source.
Socket sourceSocket;     // Socket to the event source.
DataInputStream is;       // Input stream for reading from the source.
DataOutputStream os;      // Output stream for writing to source.
MsgBuffer msgBuffer;      // Used for all communication with the source
                          // by the source thread.
MsgBuffer filterBuffer;   // Used to send idNewSelection messages
                          // to the source.
                          // These messages are sent by client threads
                          // so MsgBuffer cannot be used.
EvtQ evtQ;                // This thread puts event messages into this
                          // queue after they are received from the source.

static String VIEW_NAME = "evtforward";
static String fullName = null;// This string is used as sendDsp in messages
                          // to source. Format is host>port.
```

SourceThread methods:

SourceThread()

This constructor allocates `msgBuffer` and `filterBuffer`. It also creates `noDataWarning`, the message sent to all client views if we lose contact with the event source.

boolean **connectToSource**(boolean printError)

This method is called only when `protectedSource` is false. It connects to the server at sourceAddr, sourcePort. While doing that it sets fields `sourceSocket`, `is` and `os`. It returns true if all goes well. False otherwise. The `evtforward` constructor calls this method before calling this class's run method to allow better error handling if the connection fails.

public void **run**()

This method does the following:

1.

   Creates the String `fullName`, consisting of the name of the host and the port used by this process. This string is used as the name of the sending display in messages to the event source. Including the port number in the string allows multiple event processes to run on the same host. For example, an instance of dsp_evtdsp could run on the same host as this evtforward.
2.

   If sourceSocket is null, `reconnect` is called. This happens when -wait was specified on the command line and the event source had not opened its socket when main called `reconnect`. Note that `reconnect` calls `requestView` after it reconnects.

   If sourceSocket is non-null, the socket is open and the only action is to call `requestView`. This requests one view from the event source.
3.

   Waits for messages from the source. If an event message is received, it is passed to method `processEvent`. Other types of message are ignored. If contact with the event source is lost, method `reconnect` is called to try to reestablist contact.

private boolean **requestView**()

This method is called when a new connection is established with an event source. The method sends an idGetView message to the source requesting a view named VIEW_NAME with a filter equal to ClientList.`filterUnion`.

During the initial connection, `filterUnion` is zero, meaning that no events are sent to this view. If the connection is broken and re-established at a later time, there may be clients so `filterUnion` is non-zero.

If the message is successfully sent, true is returned. False otherwise.

protected void **processEvent**()

This method is called after an event message has been put into msgBuffer. It does the following:

1. Locates the event text.
2. Determines the event number.
3. Calls EvtQ.`put` to put the event text and number into the event queue.

private void **reconnect**(boolean closedOK)

This method is called when the connection to the event source is lost. It does not return until the connection is reestablished.

Its normal action is to call method `connectToSource` every `waitMillis` milliseconds until that method succeeds. However, if `protectedSource` is true, it waits for the event source to connect to `dataSocket`.

After success, method `requestView` is called.

If `closedOK` is false, a message is sent to all client views informing them that connection to the source has been lost. When the connection is restored, another message is sent.

private void **closeSocket**()

This method tests `sourceSocket`. If it is non-null, the method closes `sourceSocket`, `is` and `os`.

synchronized static void **sendNewFilter**()

This method is called by ClientList.`checkFilterUnion` after ClientList's filterUnion has changed. It sends an idNewSelection containing the new filterUnion to the event source.

If `sourceSocket` is null when this function is called, the connection to the event source is broken and nothing is done. The new union will be requested when connection is re-established.

This method is synchronized because it runs in a client thread and it tests `sourceSocket` which is set by the source thread. It builds the idNewSelection message in the buffer `filterBuffer`, which is used only in this method. `msgBuffer`, used for other communications with the source, cannot be used here because the source thread may be using it.

3.13 Class MsgBuffer

This class is used by java applications and applets that send or receive messages in the evtdsp message format. At the moment that includes EvtdspApp and evtforward.

An object of this class contains a buffer where messages can be constructed before being being sent and into which messages from other processes are read. Class methods take care of inserting and removing wrappers. Other methods insert strings, integers and filters into the buffer and retrieve them from the buffer.

The only public data member in the class is the String **errorString**. Several methods put a message into this member when they return an error value.

Protected data fields:

```
byte[] buf;     // The buffer where messages are constructed before
                // being sent and where received messages are put.
                // Wrappers are put here along with message content.
int bufSpace;   // The number of bytes currently allocated in buf.
int contentBytes; // The number of message content bytes currently
                // in buf.
int readIndex; // The index in buf of the next byte to be removed
                // by methods nextString, getInt and getFilter.
```

The constructor can optionally take an integer buffer length.

Interface functions:

3.14 void empty()

This method is called when the user wants to begin constructing a new message in the buffer.

The method resets contentBytes to 0.

3.15 void setBufSpace(int bufLen)

This method insures that bufSpace, the number of bytes currently allocated for the buffer, is at least `bufLen`. If contentBytes, the number of content bytes in buf, is greater than 0, those bytes are copied into the new buffer.

3.16 void appendString(String str, boolean addNull)

This method appends `str` to the buffer. If `addNull` is true, a null character is appended after `str`.

If necessary, the buffer is expanded.

3.17 void appendInt(int i)

This method changes `i` to a string and calls appendString to append it to the buffer.

If necessary, the buffer is expanded.

3.18 void appendFilter(long filter)

This method appends `filter` to the buffer, low-order byte first.

If necessary, the buffer is expanded.

3.19 String nextString()

This method returns the next string from the buffer. Null is returned if the buffer does not contain any more strings. If the string is null-terminated, the null character is removed from the String returned.

The protected variable `readIndex` is moved past the string returned, so the next call to this method will return the next string from the buffer.

3.20 int getInt()

If the next string in the buffer consists only of decimal digits, this method returns the string's integer value.

If there is a problem, -1 is returned. At the moment, this method is used only to process message type fields, which are never negative, so this return is not ambiguous.

3.21 FilterReturn getFilter()

The object returned by this function consists of two data fields:

```
long filter;
boolean filterOK;
```

If there are fewer than four content bytes remaining in the buffer, the `filterOK` field of the object returned is set to false.

Otherwise `filterOK` is set to true and the next four bytes are converted to a long value and assigned to `filter` in the object returned. The first byte of the four is put into the low order byte of `filter`.

3.22 int sendMsg(DataOutputStream os)

This method creates header and footer fields (see Message wrappers) for the current contents of the buffer and writes it to `os`.

If the write is successful, 0 is returned. If it looks like `os` is no longer open MsgBuffer.CLOSED is returned. Otherwise MsgBuffer.ERROR is returned and an error message is put into errorString.

3.23 int recvMsg(DataInputStream is)

This method waits until a message is available on `is`. The message is put into the buffer and the instance variable `contentBytes` is set to the number of content bytes read, and the instance variable `readIndex` is set so the first string in the message will be returned by the next call to nextString(), getInt() or getFilter().

If it looks like `is` is no longer open, MsgBuffer.CLOSED is returned. If anything else goes wrong, MsgBuffer.ERROR is returned and a message is put into errorString. If all goes well, contentBytes is returned.

4 Message formats

Three nested levels of formatting are used in event processing:

Outside connections gives information on how other systems can receive ITOS events.

4.1 Event messages

This section describes the event messages sent by dsp_evtlog to dsp_evtdsp via a pipe. When these messages are sent over sockets between dsp_evtdsp, evtforward and EvtdspApp, they are contained in idEvent messages.

The event messages sent by dsp_evtlog to dsp_evtdsp can currently start with zero or more of the following escape sequences:

- Esc [0 m = Reset. Set foreground to fontColor7 and background to fontColor0.
- Esc [3 0 m = Set foreground to fontColor0.
- Esc [3 1 m = Set foreground to fontColor1.
- Esc [3 2 m = Set foreground to fontColor2.
- Esc [3 3 m = Set foreground to fontColor3.
- Esc [3 4 m = Set foreground to fontColor4.
- Esc [3 5 m = Set foreground to fontColor5.
- Esc [3 6 m = Set foreground to fontColor6.
- Esc [3 7 m = Set foreground to fontColor7.
- Esc [4 0 m = Set background to fontColor0.
- Esc [4 1 m = Set background to fontColor1.
- Esc [4 2 m = Set background to fontColor2.
- Esc [4 3 m = Set background to fontColor3.
- Esc [4 4 m = Set background to fontColor4.
- Esc [4 5 m = Set background to fontColor5.
- Esc [4 6 m = Set background to fontColor6.
- Esc [4 7 m = Set background to fontColor7.

The escape sequences are optionally followed by two decimal digits specifying the event type. If those digits are missing, event type 0 is assumed.

The event text follows. It is terminated by a linefeed character.

4.2 Evtdsp messages

This section describes messages passed between dsp_evtdsp, evtforward and EvtdspApp. It also describes messages passed between a dsp_evtdsp and its child dsp_remote.

Note that before any of these messages are sent over a socket, a header and footer are wrapped around them as described in Message wrappers.

All messages begin with an integer that identifies the message type. Symbolic constants beginning with "id" are used for those integers in source code and in this section.

These constants are defined in '`evt_remote.h`' for dsp_evtdsp and in '`MsgBuffer.java`' for evtforward and EvtdspApp.

That message type is put into a message as a null-terminated string of ascii decimal digit characters. Dsp_evtdsp and dsp_remote always use two digit characters, with the first possibly being '0'. EvtdspApp does not insert the leading '0' and so may use only one digit character.

Some of the messages described below contain the field <filter>. That is an eight-byte bitfield specifying the set of events selected for a view. The first byte is for events 1-8, the next is for 9-16, etc. The least significant bit in each byte is for the lowest-numbered event in that byte.

All fields except <filter> are null-terminated text strings.

In the following message descriptions, <sendProc> identifies the process sending the message. dsp_evtdsp typically uses the name of the host where it is running. If it is using a non-standard port, it appends ">num" to the host name, where "num" is its port number. Evtforward always uses this "host>num" format. EvtdspApp uses "applet" followed by a number based on the time the applet (or application) was started.

The various <sendProc> formats described above are needed because all processes communicating with a given process must use different strings in their <sendProc> fields.

<recvProc> is the name of the process which will receive the message.

The messages are:

- idCloseDisplay <process>
  - Sent from dsp_evtdsp to dsp_remote when the user deletes a remote display. dsp_remote does not send any message to <process>, but closes its socket to <process>.
  - Sent from dsp_remote to dsp_evtdsp when dsp_remote fails to connect to a process. (See idOpenDisplay.)
- idCloseView <sendProc> <recvProc> <view name>
  - Sent from dsp_evtdsp to dsp_remote when the user closes a view on a remote display.
  - dsp_remote forwards the message to <recvProc>.
  - Sent by EvtdspApp every time the user closes a view.
- idDebug [<text>]
  - Sent from dsp_evtdsp to dsp_remote to tell dsp_remote to start sending debug messages to dsp_evtdsp. There is no text with this message. dsp_evtdsp sends this message when the debug window __debug__ is opened.
  - Sent with text from dsp_remote to dsp_evtdsp for display in the debug window __debug__.
- idError <text>
  - Sent from dsp_remote to dsp_evtdsp when an error occurs. See also idUError.
- idEvent <sendProc> <recvProc> <event text> <view name> ...

  - Sent by dsp_evtdsp and evtforward when they receive an event required by a remote client. Evtforward sends the message directly to <recvProc>, dsp_evtdsp sends it to its dsp_remote, which forwards it to <recvProc>.
  - <event text> is the complete text of an event message, ending with a linefeed character. In this message, a null character is placed after that linefeed.
  - The <event text> field is followed by one or more <view name> fields. Each is a null-terminated string naming a view on <recvProc>.
- idGetView <sendProc> <recvProc> <view name> <filter>
  - Sent by dsp_evtdsp when the user wants to open a view showing events from a remote display. The dsp_remote of the sending dsp_evtdsp forwards the message to <recvProc>. That may require opening a socket to <recvProc>.
  - Sent by EvtdspApp to its event source every time the user opens a new view.
  - Sent by evtforward to its event source when it connects to that source.
  - <filter> specifies the events that will be displayed in <view name>. The structure of <filter> is described near the start of this section.
- idNewSelection <sendProc> <recvProc> <view name> <filter>
  - Sent by dsp_evtdsp when the user changes the events selected for a view displayed by <recvProc>, or a view displaying events obtained from <recvProc>.
  - Evtforward sends one of these messages to its event source when the union of all filters of all of its clients changes.
  - Sent by EvtdspApp every time the user changes the filter of any view.
  - <filter> specifies the new set of events to be displayed in <view name>. The structure of <filter> is described near the start of this section.
- idNoDebug
  - Sent from dsp_evtdsp to dsp_remote to tell dsp_remote to stop sending debug messages. Dsp_evtdsp sends this message when the user closes the __debug__ view.
- idOpenDisplay <process>
  - Sent from dsp_evtdsp to dsp_remote. If dsp_remote does not currently have a socket open to <process>, it tries to open one.
  - If dsp_remote is successful (or if the socket is already open), it returns the message to dsp_evtdsp.
  - If dsp_remote is not successful, it returns an idCloseDisplay <process> message to dsp_evtdsp.
- idQuit
  - Sent from dsp_evtdsp to dsp_remote to tell it to exit.
- idSendView <sendProc> <recvProc> <view name> <filter>
  - Sent from dsp_evtdsp to dsp_remote when the user wants to send data to a remote display.
  - Dsp_remote forwards the message to <recvProc>. That may require opening a socket to <recvProc>.

  - <filter> specifies the set of events to be displayed in <view name>. The structure of <filter> is described near the start of this section.
- idUError <text>
  - Sent from dsp_remote to dsp_evtdsp.
  - Dsp_evtdsp displays <text> to the user in a dialog.
- idEnsureDelivery <blockCount>
  - Sent from a remote user to ITOS host port 6066. The user is requesting that ITOS buffer outgoing events if necessary to avoid dropping events. A total of blockCount 1024 byte blocks will be used as buffers to accomplish this function
  - Once a reliable connection has been established subsequent idEnsureDelivery messages on this port are ignored.

4.3 Message wrappers

This section describes the header and footer wrapped around every evtdsp message before it is sent over a socket.

The header consists of a zero-padded, null-terminated ascii string giving the length of the message in decimal. The header is not counted in this length, but the footer is.

The length of this header is given by the constant REMOTE_HEADER_LEN in '`dsp_remutil.c`' and by the constant MsgBuffer.HEADER_LEN in '`MsgBuffer.java`'. Those constants must match and are currently defined to be 5. That length includes the header's null-terminator.

The header is followed by the message content as described in Evtdsp messages.

The last byte of message content is followed by a null-terminated footer string, defined as REMOTE_FOOTER_STR in '`dsp_remutil.c`' and as MsgBuffer.FOOTER in '`MsgBuffer.java`'. Those string definitions must match and are currently defined to be "end".

5 Outside connections

This section gives information useful to those developing non-ITOS systems which receive ITOS event messages. It contains links to relevant sections of ITOS documentation, followed by additional information needed by outside developers.

SOCKETS

Event messages are sent over a Tcp/Ip socket. You'll have to get the server host name and port number from operations personnel. Typically two ITOS processes open event server sockets on each machine:

> "evtforward" and uses port 6067 by default and is the recommended server. It is often running when ITOS is shut down. If that's the case, it will send a null event message saying so. It will automatically connect to ITOS when it starts up and you will begin receiving messages.
>
> If evtforward is not running, you can use "dsp_evtdsp". That server uses port 6066 by default and runs only when ITOS is running.

Message wrappers describes the outermost layer of messages.

Messages specifies the fields in all messages. The text below tells which messages are useful to you, but assumes you'll get details from Messages.

MESSAGES YOU SEND TO ITOS

To receive event messages, send the idGetView message. Its fields can be set as follows:

- idGetView = "06\0"
- <sendProc> = Your host's name.
- <recvProc> = The name of the ITOS host.
- <view name> = Any non-empty, null-terminated string.
- <filter> = 0xffffffff requests all events. To be more selective, build a bitfield. Event id numbers are given at the bottom of this page. (Messages gives bitfield details.)

The "view" field is present because ITOS processes can open several windows, each displaying a different set of events. Each window is a "view", and each event message you receive names the views in which the event should be displayed.

You can send multiple idGetView messages with different <view name> fields (and usually different filters).

To close a view, send an idCloseView message. idCloseView = "02\0".

To break your connection to ITOS, simply close your end of the socket. When ITOS is shut down, it will close its end of the socket.

To change the events displayed in an existing view, send an idNewSelection message. idNewSelection is "07\0" and other fields are the same as in idGetView.

MESSAGES YOU RECEIVE FROM ITOS

Event messages start with idEvent ("05\0"). Notice that the text of the event (described in Event message text) is preceded by the name of the sending host and the name of your host. It is followed by the view names you have used in idGetView and idNewSelection messages requesting the event.

It is unlikely, but possible, that you will receive the following three messages, especially if you connect to dsp_evtdsp. Your connection is visible to ITOS operators and they can delete your views, add views for you and change your filters. (Possibly by mistake.)

idCloseView "02\0". This means you won't get any more events on the view named.

idSendView "12\0". This means you will start getting events naming this new view.

idNewSelection "07\0". This means the view named will get a different set of events in the future.

Event message text describes the innermost part of the event messages you will receive. The following symbolic names are used for colors on that page:

- fontColor0 = black
- fontColor1 = red
- fontColor2 = green
- fontColor3 = yellow
- fontColor4 = blue
- fontColor5 = magenta
- fontColor6 = cyan
- fontColor7 = white

That page refers to digits specifying event types. They are:

- 00 = Null event
- 01 = Red limits violation
- 02 = Yellow limits violation
- 03 = Delta limits violation
- 04 = Value back in limits
- 05 = Telemetry information
- 06 = Telemetry warning
- 07 = General telemetry warning
- 08 = command event
- 09 = Command verify/no-verify event
- 10 = Configuration error message
- 11 = Command informational message
- 12 = Command warning message
- 13 = General command error message
- 14 = Command transfer frame echoed in hex
- 15 = STOL Operator error
- 16 = STOL echo of directives
- 17 = STOL MSG directive message
- 18 = STOL warning message
- 19 = STOL error message

- 20 = Display informational message
- 21 = Display warning message
- 22 = Display error message
- 23 = FTCP xmit proc informational message
- 24 = FTCP xmit proc warning message
- 25 = FTCP xmit proc error message
- 26 = SunOS Kernel message (not TCW s/w message)
- 27 = Serious error message – call a programmer!
- 28 = Spacecraft event message
- 29 = Configuration monitor alert message
- 30 = A debugging message
- 31 = Science Data Processing message
- 32 = Science Data Processing warning
- 33 = Science Data Processing error
- 34 = Process Controller message
- 35 = Process Controller warning
- 36 = Process Controller error
- 37 = CFDP Driver message
- 38 = CFDP Driver warning
- 39 = CFDP Driver error
- 40 = Not currently used.
- 41 = Not currently used.
- 42 = Not currently used.
- 43 = Not currently used.
- 44 = Not currently used.
- 45 = Not currently used.
- 46 = Not currently used.
- 47 = Not currently used.
- 48 = Not currently used.
- 49 = Not currently used.
- 50 = Not currently used.
- 51 = Not currently used.
- 52 = Not currently used.
- 53 = Not currently used.
- 54 = Not currently used.
- 55 = Not currently used.
- 56 = Not currently used.
- 57 = Not currently used.

- 58 = Not currently used.
- 59 = Not currently used.
- 60 = Not currently used.
- 61 = Not currently used.
- 62 = Not currently used.
- 63 = Not currently used.
- 64 = Not currently used.